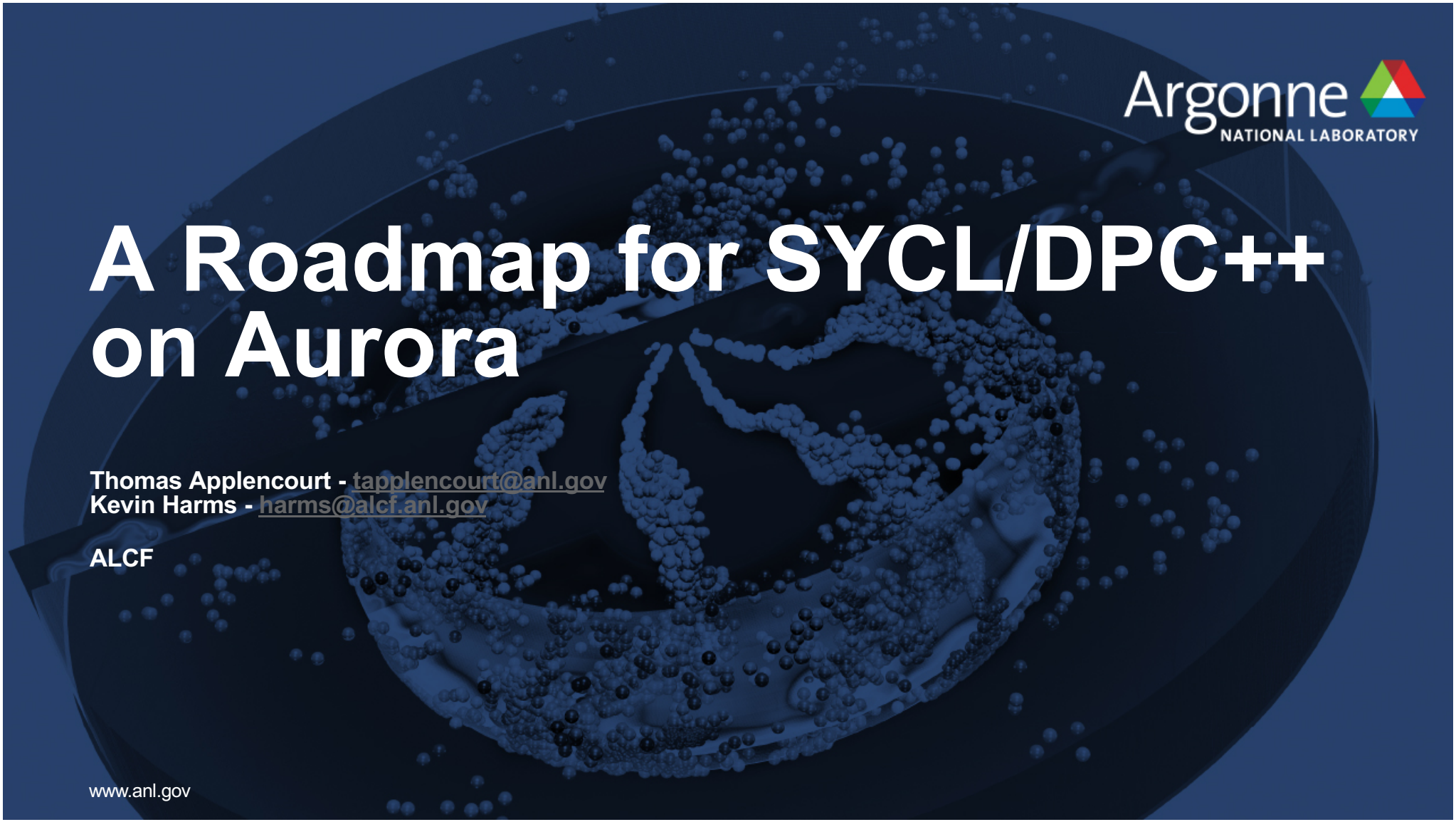


Argonne NATIONAL LABORATORY

A Roadmap for SYCL/DPC++ on Aurora

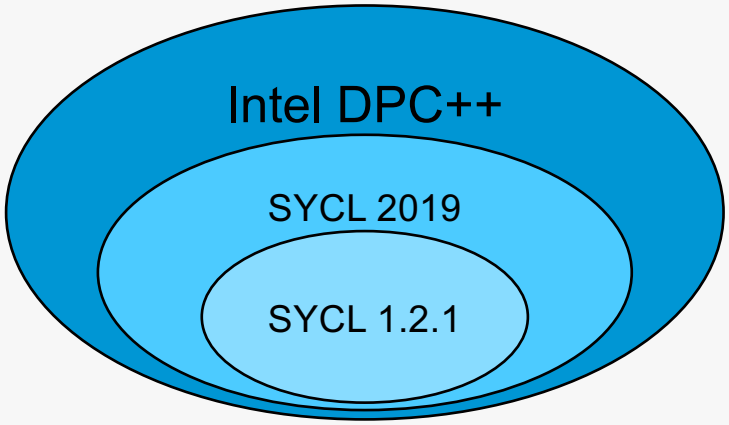
Thomas Applencourt - tapplencourt@anl.gov
Kevin Harms - harms@alcf.anl.gov

ALCF

DPC++ (Data Parallel C++) and SYCL

Overview

- SYCL
  - *SYCL (pronounced "sickle") is a royalty-free, cross-platform abstraction C++ programming model for OpenCL. SYCL builds on the underlying concepts, portability and efficiency of OpenCL while adding much of the ease of use and flexibility of single-source C++*
  - Khronos standard specification
  - *SYCL is designed to be as close to standard C++ as possible*
- Data Parallel C++ (DPC++)
  - Intel interpretation/implementation of the SYCL specification
  - Incorporates SYCL 2019 specification and Unified Shared Memory
  - Add language or runtime extensions as needed to meet user needs



Intel oneAPI

- oneAPI is broad-based software ecosystem that works together to accelerate software on multiple different device architectures
  - DPC++ is one component in that ecosystem

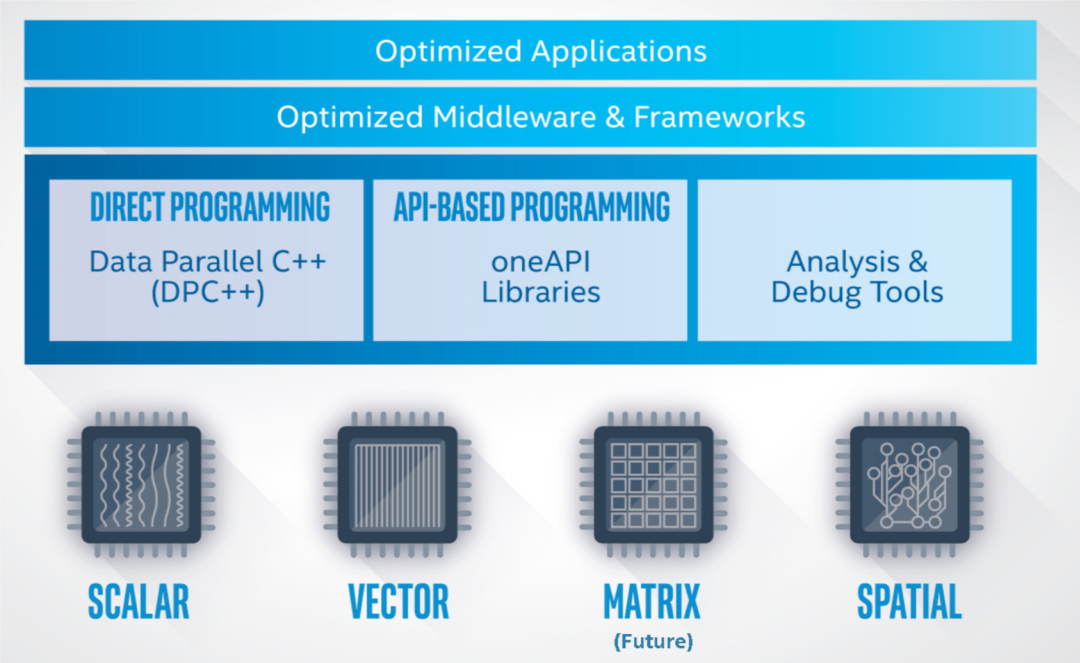


Image courtesy of Intel

Why SYCL?

Who should listen to this presentation...

You ...

- are creating a greenfield application or complete rewrite and chose C++ as your base language
- are coming from CUDA or HIP and you want to embrace an open standard

Overview

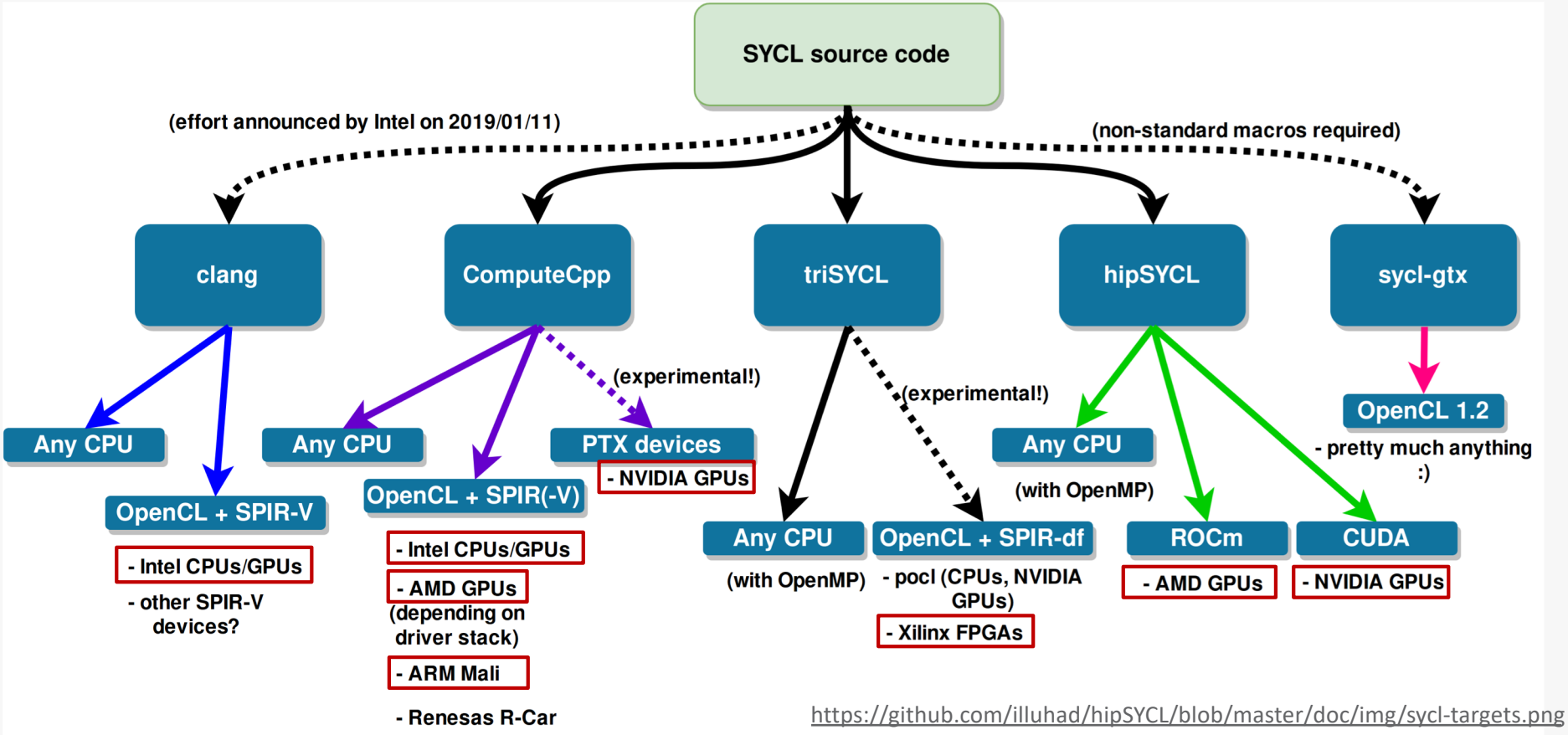
... royalty-free, cross-platform abstraction layer that builds on the underlying concepts, portability and efficiency of OpenCL that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++. SYCL single-source programming enables the host and kernel code for an application to be contained in the same source file, in a type-safe way and with the simplicity of a cross-platform asynchronous task graph. SYCL includes templates and generic lambda functions to enable higher-level application software to be cleanly coded with optimized acceleration of kernel code across the extensive range of shipping OpenCL 1.2 implementations.

-- SYCL Specification 1.2.1

- standard C++11
- OpenCL 1.2 memory model

- based on OpenCL - reuses the same concepts for definition of the architecture
  - platform, device, queue, NDRange, work groups and work items
- this presentation is just scratching the surface of SYCL

SYCL Ecosystem



https://github.com/illuhad/hipSYCL/blob/master/doc/img/sycl-targets.png

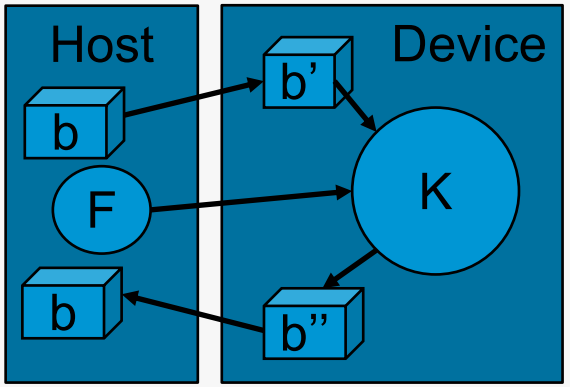
Foreword

- SYCL is based modern C++ (standard C++11)
- We will review C++ concepts and syntax as we go
- You will need a good grasp of certain C++ syntax to understand the code examples

Platforms and Devices

- Platforms contain devices
  - A host may have multiple platforms
  - A platform may have multiple devices
- Platforms are loosely defined as an implementation runtime
- Devices correspond to some hardware that the SYCL code will run on
- Iris/Gen9 nodes
  - 3 platforms, each with one device
  - Host, CPU, GPU

Pseudo-code walkthrough




```
selector = default_selector()
q = queue(selector)
b = buffer (double, 1000)
q.submit (
      F ( ) {
            a = accessor(b, READ_AND_WRITE)
            K (i) {
                  a[i] = a[i] * 2
            }
      }
)
q.wait_and_throw()
a2 = accessor(b, READ_ONLY)
printf("%f", a2[0])
```

C++ Review - const

- *const* has been around in C for many years
- modern C++ compilers are more strict about const correctness
- If your code looks right but you have an odd error about the LHS/lvalue or RHS/rvalue, could be due to a const correctness issue

const int

```
file.cpp:40:28: error: non-const lvalue reference to type
'sycl::device_selector' cannot bind to a temporary of type
'sycl::cpu_selector' sycl::device_selector &selector_cpu =
sycl::cpu_selector();
```

Device Selection

- SYCL provides the concept of a device selector which is a function that will choose which device to run on
- device selectors select that type of device

- **sycl::default_selector**
  - implementation specific method to select a default device
- **sycl::cpu_selector**
  - selects the host device
  - The host processor runs the OpenCL implementations and is a single or multi-core CPU.
- **sycl::gpu_selector**
  - selects a GPU device
  - can be used to accelerate a 3D API such as OpenGL or DirectX.
- **sycl::accelerator_selector**
  - selects an accelerator device
  - communicate with the host processor using a peripheral interconnect such as PCIe.

Device Selection Example

```
using namespace cl;

sycl::device_selector selector = sycl::gpu_selector();

sycl::queue queue(selector);
```

- This will select a GPU device
- If a GPU device doesn't exist, queue creation will fail.

C++ Review - Template functions

- Template functions similar to Template classes
- Provide generic types for functions

```
template <typename T> void func (T a)
      { T b = a; return; }
func<int>(4);
```

Device queries using get_info():

Descriptor in info::device	Return type
device_type	info::device_type
vendor_id	cl_uint
max_compute_units	cl_uint
max_work_item_dimensions	cl_uint
max_work_item_sizes	id<3>
max_work_group_size	size_t
preferred_vector_width_char	
preferred_vector_width_short	
preferred_vector_width_int	
preferred_vector_width_long_long	cl_uint
preferred_vector_width_float	

```
using namespace sycl;
template <info::device param> typename info::param_traits<
info::device, param>::type get_info() const;

cl_uint eu = device.get_info<info::device::max_compute_units>();
```

Tutorial 1

- Introduce test platform
- Build environment and tools
- Options for testing
  - Container
  - JLSE
- First example

Queue Management

- Queue constructor with the device selector binds to the queue to the device
  - can also be initialized with a specific device
- kernels submitted will run asynchronously
- *submit* - takes a function object to execute
  - runs host code portion synchronously
  - once kernel submitted to device, code runs asynchronously
  - errors during *submit()* or *wait()* reported synchronously via exceptions
- errors that occur after kernel submission are reported asynchronously and require defining an async_handler
  - *wait_and_throw* or *throw_asynchronous* must be called to run the handler
- queue object destructors will wait for queued kernels to complete before the destructor completes

Queue Management Example

```
sycl::queue q(selector);

q.submit(...);
q.wait() // discards all exceptions
// q.wait_and_throw();

// implicit synchronization on queue destructor
{
     sycl::queue q(selector);
}
```

Tutorial 2

- First Kernel

Kernels

- Three types of kernels
  - **single_task**
    - run one instance of the kernel, no local accessors
  - **parallel_for**
    - run number of instances based on workgroups and workitem
    - number of variants for different types of arguments
  - **parallel_for_work_group**
    - allows similar capability as with nd_range
    - can run workgroup code that runs only once for the workgroup
    - can allocate local memory and private memories
    - use **parallel_for_work_item** within to parallelize over work items

C++ Review - Templates

- the best (or the worst) thing about C++
- allow creating code that uses a generic type
- Removes the need to write many virtual functions with different types

```
template <typename T>
class vec3 {
     T val[3];
     public:
          vec3 (T x, T y, T z) {
               val[0] = x; val[1] = y; val[2] = z;
          }
};
vec3<double> a(1.0, 2.0, 3.0);
vec3<int> b(1, 2, 3);
```

C++ Review - lambda

- lambda is common in many languages, same concept in C++
- creates an anonymous function that can capture variables in scope
- *[capture] (parameters) { body }*
  - [] capture nothing
  - [&] capture by reference
  - [=] capture by making a local copy
  - [a, &b] capture explicit variables

```
void func (std::function<void(int)> f) { int x = 6; f(x); }

{
  int a = 6;
  func([&](int q){ if (q == a) std::cout << "success"; });
}
```

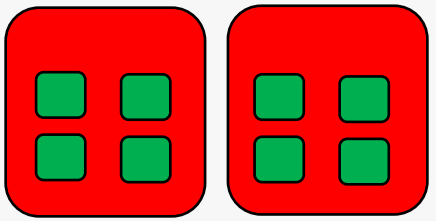
Kernel - single_task

```
using namespace cl;

queue.submit(
      [&](sycl::handler &cgh)
      {

            cgh.single_task<class kernel1>(
                  [=]() // no arguments
                  {
                        ...
                  }
            );
      }
);
```

Kernel - parallel_for

```
using namespace cl;
sycl::buffer<int, 1> val(sycl::range<1>(100));
queue.submit(
      [&](sycl::handler &cgh)
      {
            auto a = val.get_access<sycl::access::mode::write>(cgh);
            cgh.parallel_for<class kernel2>(sycl::range<1>(100),
                  [=](sycl::id<1> i)
                  {
                        size_t ii = i[0];
                        a[ii] = 10;
                  }
            );
      }
);
```

Kernel - parallel_for_work_group



```
using namespace cl;
sycl::buffer<int, 1> val(sycl::range<1>(8));
queue.submit(
  [&](sycl::handler &cgh)
  {
    auto a = val.get_access<sycl::access::mode::write>(cgh);
    cgh.parallel_for_work_group<class kernel3>(
      sycl::range<1>(2), sycl::range<1>(4),
      [=](sycl::group<1> g)
      {
        int workgroup_local = g.get_id(0);
        g.parallel_for_work_item(
          [&](sycl::h_item<1> item)
          {
            size_t ii = item.get_global_id();
            a[ii] = workgroup_local;
          });
      });
  });
```

Range Operators

- SYCL provides seven classes to handle expressing data decomposition
  - range, nd_range
  - id, item, nd_item
  - group, h_item
- range<dimensions>(<size of dimension>)
  - `range<1>(200)`
  - `range<2>(4, 2)`
- id<dimensions> - provides the index into the range
  - `id<1> a; size_t index = a[0];`
  - `id<2> b; size_t x = b[0]; size_t y = b[1];`
- nd_range<dimensions>(range<dimension> global_size, range<dimension> local_size)
  - `nd_range<1>(range<1>(256), range<1>(128))`
- nd_item<dimensions> - provides index into the nd_range plus more functionality
  - `nd_item<1> a;`
  - `size_t global_index = a.get_global_id(0);`
  - `size_t local_index = a.get_local_id(0);`

Range Example

```
using namespace cl;

q.submit([&](sycl::handler &cgh)
  {

    cgh.parallel_for<class kernel>(sycl::range<1>(4096),
        [=](sycl::id<1> x)
        {
          size_t i = x[0];
        }
    );
  }
);
```

Tutorial 3

- Basic parallel_for example

ND_Range Example

```
using namespace cl;
sycl:buffer<int, 1> buf(cl::sycl::range<1>(4096));
q.submit([&](cl::sycl::handler &cgh)
  {
    auto acc = buf.get_access<sycl::access::mode::discard_write>(cgh);
    cgh.parallel_for<class kernel>(
      sycl::nd_range<1>(sycl::range<1>(4096), sycl::range<1>(64)),
      [=](sycl::nd_item<1> ndi)
      {
        size_t i = ndi.get_global_linear_id();
        acc[i] = i;
      });
});
```

Tutorial 4

- Parallel_for with nd_range

Buffer Management

- SYCL provides host managed memory in the form of buffer or image
- sycl::buffer
  - manages copying data back and forth between host and device
  - can be constructed with cl_mem object, host data, or allow SYCL to allocate
  - use accessors to get access to data (next topic)
- sycl::image
  - special instance of buffer to support image concepts such as channel order and image format
- explicit copying is possible via *copy* interface provided by the sycl::handler class
- private memory local to a *workitem* can be allocated inside of
  - parallel_for or parallel_for_work_item
- semi-private memory local to a *workgroup* can be allocated inside of
  - parllel_for_work_group
  - accessor using sycl::local_accessor
  - Shared Local Memory or SLM

Buffer Management Example

```
template <typename T, int dimensions = 1, typename AllocatorT =
sycl::buffer_allocator> class buffer;

buffer(const range<dimensions> &bufferRange, const property_list &propList =
{});
```

```
// create buffer from existing memory allocation
double *data = malloc(sizeof(double) * 1000);
data[0] = 1.0;
sycl::buffer<double, 1> buf1(data, sycl::range<1>(1000));

// SYCL handles memory allocation and deallocation
sycl::buffer<double, 1> buf2(sycl::range<1>(1000));
```

C++ Review - auto

- keyword that can be used to replace any type declaration
- compiler will determine what type the variable is based on what the variable is being initialized with
- eases the burden for the writer of the code
- raises the burden for the reader of the code
- good use is with accessor variables

```
std::map<std::pair<std::string,std::string>,std::vector<int>>::iterator iter;
iter = mymap.begin();

auto iter = mymap.begin();
auto i = 0x5;
```

Accessors

- accessors are used to access memory from buffers, images or local memory
- inform runtime of your intent to access data and allows runtime to schedule access
- synchronize access to data
- accessor has five components
  - data type (int, float, etc.)
  - dimensions (1, 2, 3)
  - access mode (read, write, read_write, discard_write, discard_read_write, atomic)
  - target (global_buffer, constant_buffer, local, host_buffer)
  - placeholder (defaults to false)

Accessor Example

```
using namespace cl;
sycl::buffer<double, 1> buf2(sycl::range<1>(1000));

queue.submit(
  [&](sycl::handler &cgh) {
    auto b_acc = buf2.get_access<sycl::access::mode::read_write>(cgh);
    cgh.parallel_for<class kernel>(sycl::range<1>(1000),
        [=](sycl::id<1> x)
        {
          double vv = 5.0; b_acc[x[0]] = vv;
        });
  });

auto b_host_acc = buf2.get_access<sycl::access::mode::read_write>();
// use b_host_acc on host side to allow runtime to synch access
// will wait until data is copied back to host
double v = b_host_acc[5];
```

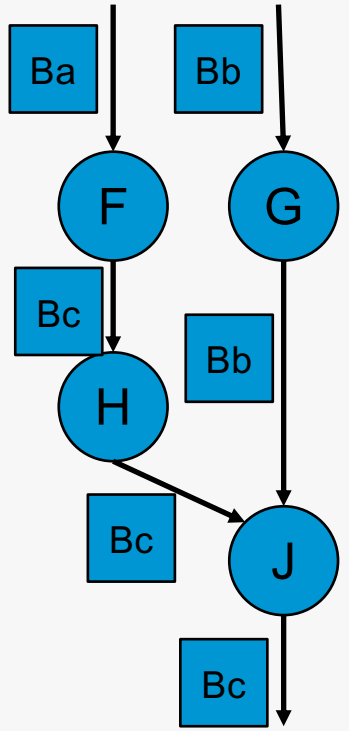
Thomas 5

- Parallel_for with buffer and accessor
- Buffer with shared local memory (SLM)

Accessor DAG

- A set of accessors will construct a DAG which the runtime will understand
- This will allow runtime to schedule work in optimal fashion based on data access rules

```
sycl::buffer<float, 1> Ba (sycl::range(4096));
sycl::buffer<float, 1> Bb (sycl::range(4096));
sycl::buffer<float, 1> Bc (sycl::range(4096));
```



Accessor DAG Example

```
queue.submit([&](sycl::handler &cgh) {
      auto Aa = Ba.get_access<sycl::access::mode::read>(cgh);
      auto Ac = Bc.get_access<sycl::access::mode::discard_write>(cgh);
      ...<kernel_F>... });

queue.submit([&](sycl::handler &cgh) {
      auto Ac = Bc.get_access<sycl::access::mode::read_write>(cgh);
      ...<kernel_H>... });

queue.submit([&](sycl::handler &cgh) {
      auto Ab = Bb.get_access<sycl::access::mode::read_write>(cgh);
      ...<kernel_G>... });

queue.submit([&](sycl::handler &cgh) {
      auto Ab = Bb.get_access<sycl::access::mode::read>(cgh);
      auto Ac = Bc.get_access<sycl::access::mode::read_write>(cgh);
      ...<kernel_J>... });
```

Tutorial 6

- Kernel dependency with buffer/accessor DAG

Unified Shared Memory (sycl::malloc)

- Not currently part of SYCL 1.2.1 spec and not in June SDK
- a way to use SYCL without buffers and accessors
- requires certain properties of memory access/visibility between host and device
- https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/USM/USM.adoc

```
void* sycl::malloc_device (size_t size,
                           const sycl::device& dev,
                           const sycl::context& ctxt);

void* sycl::malloc_shared (size_t size,
                           const sycl::device& dev,
                           const sycl::context& ctxt);

void sycl::free (void* ptr, sycl::context& context);
```

Tutorial 7

- Unified Shared Memory example

SYCL Implementations

- Intel LLVM SYCL
  - https://github.com/intel/llvm
- Codeplay ComputeCpp
  - https://www.codeplay.com/products/computesuite/computecpp
- triSYCL
  - https://github.com/triSYCL/triSYCL
- hipSYCL
  - https://github.com/illuhad/hipSYCL
- sycl-gtx
  - https://github.com/ProGTX/sycl-gtx

References

- SYCL Specification
  - https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
- SYCL Reference Sheet
  - https://www.khronos.org/files/sycl/sycl-121-reference-card.pdf
- Codeplay examples
  - https://github.com/codeplaysoftware/computecpp-sdk/tree/master/samples
- Parallel Research Kernels
  - https://github.com/ParRes/Kernels
  - under Cxx11 directory
- Thomas Applencourt's Tutorial
  - https://github.com/kevin-harms/sycltrain/tree/master/9_sycl_of_hell

Summary

- SYCL follows modern C++
  - Good or bad, you decide

- SYCL lineage from OpenCL
  - shares many properties from OpenCL
  - if you like OpenCL, you'll probably like SYCL

Advanced Topics

Error Handling

- The SYCL specification states that when `wait` is called on a queue, it will discard any exceptions.
- In order to catch exceptions, one needs to
  - call `wait_and_throw`
  - define an error handler
  - initialize a queue with the error handler

```
using sycl::async_handler = \
     function_class<void(sycl::exception_list)>;
```

Error Handling Example

```
auto ah = [](sycl::exception_list elist)
  {
    for (auto &e : elist)
    {
        std::rethrow_exception(e);
    }
  }

...
sycl::queue q({sycl::default_selector()}, ah);
...
try { q.wait_and_throw(); }
catch (sycl::exception &e)
     { std::cout << e.what() << std::endl; }
```

Device Local Memory

- SYCL provides a mechanism to declare a type of shared local memory which is visible between work-items of a work-group and is present only on the device
- Two methods to declare
  - in a parallel_for_work_group region
  - using a shared accessor
- Size allocated is per work-group

```
sycl::accessor<type, dimension, sycl::access::mode,
sycl::access::target::local> slm_acc;

auto slm_acc = sycl::accessor<int, 1, sycl::access::mode::read_write,
sycl::access::target::local>(sycl::range<1>(4096), cgh);
```

Reduction

- Currently SYCL does not define a method for reductions or provide any mechanism other than an atomic memory region
  - restricted set of types - no double

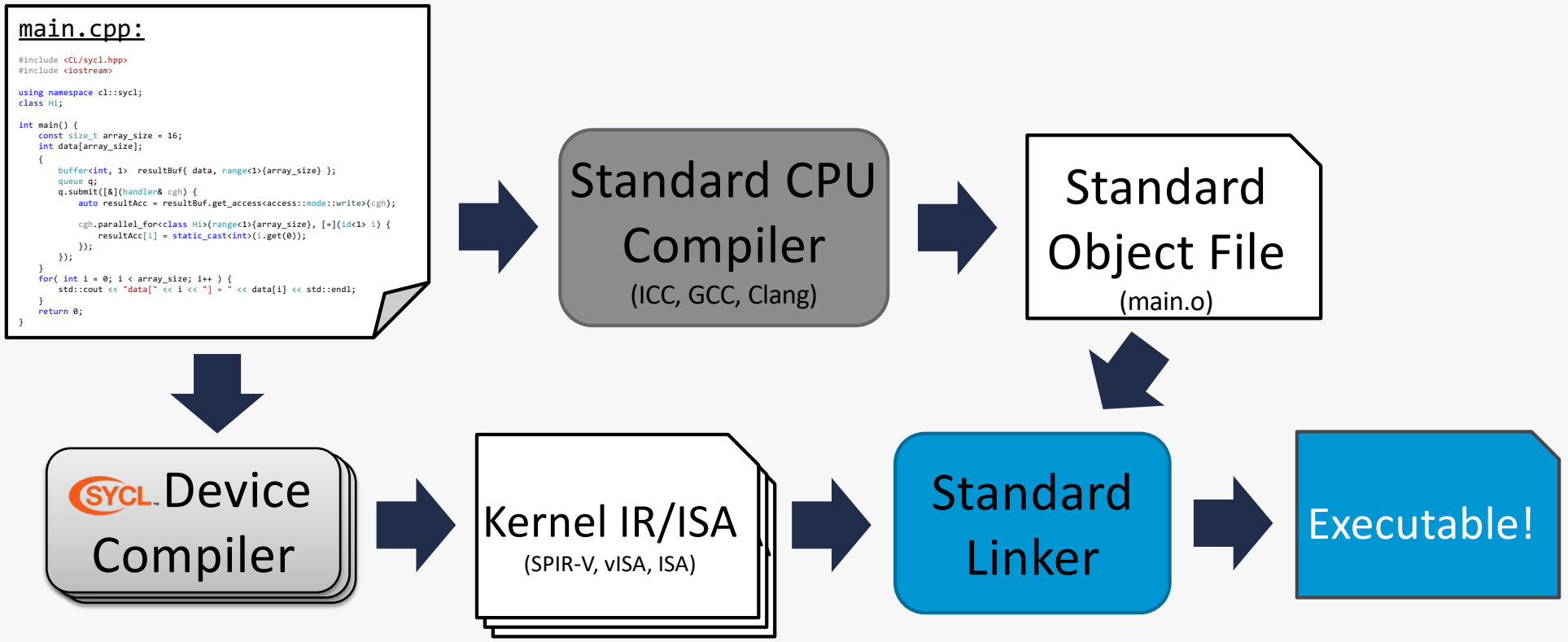
```
auto buf = sycl::buffer<int, 1>(sycl::range(1));
auto accumulated = buf.get_access<access::mode::atomic>(cgh);
accumulated[i].fetch_add(5);
```

- Intel is proposing an extension to SYCL for a reduction specific operation
  - https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/Reduction/Reduction.md

```
cgh.parallel_for<class sum>(nd_range<1>{N, M},
      reduction(span(out, 1), 0, plus<int>()),
      [=](nd_item<1> it, auto& out)
      { int i = it.get_global_id(0); out[0] += in[i]; });
```

SYCL Compiler

SYCL Compilation Flow



SYCL Execution Flow